An Analysis-Based Approach to Composition of Distributed Embedded Systems *

Pai Chou and Gaetano Borriello
Department of Computer Science and Engineering, Box 352350
University of Washington, Seattle, WA 98195-2350 USA
{chou,gaetano}@cs.washington.edu

Abstract

The growing complexity in the functionality and system architecture of embedded systems has motivated designers to raise the level of abstraction by composing the system with a mix of reusable and system-specific components. Currently, these components assume specific programming models that make them difficult to compose or retarget. The *modal process* model addresses the problem of control composition by separating the synchronization semantics from state unification, and by supporting automatic synthesis of control communication onto distributed architectures. By avoiding overspecifying the behavior, the components can be made more composable and the designer can more easily choose the least expensive synchronization semantics for implementing the composition. To help designers evaluate their choice, we propose a method for analyzing the properties of the composed system, including the detection of potential deadlock and livelock situations.

1 Introduction

Embedded systems are becoming increasingly complex. Not only is there increased functionality, but the system architecture is also becoming more complex. Specifically, *distributed* (message passing) systems have become an attractive choice because they offer good price performance tradeoffs.

The increased complexity has motivated a higher level of abstraction. The design task now is dominated by *system integration*, namely composing a set of high level components to form a complete system. A typical design would consist of a mix of mostly reusable components and some application specific components.

Currently, this component-based design approach works as long as all components to be composed are written in the same programming model. Each model makes assumptions about the synchrony of transitions, communication, and many other factors. For example, CSP [6] assumes that different processes make progress asynchronously unless they communicate by *rendezvous* synchronization. On the other extreme, the synchronous language Esterel [1] assumes that transitions are taken synchronously, as are the events.

Synchronous (event or transition) semantics have many advantages, including composability and determinism, which enable designers to succinctly and precisely capture complex control-dominated behavior. Synchronous models can be implemented efficiently on a uniprocessor. However, they can be prohibitive on distributed architectures if they have to synchronize on every transition.

Heterogeneous programming models are important for both specification and implementation. Certain applications are more naturally written in specific models (*e.g.* dataflow), which also enable their analysis and optimization. Another reason for heterogeneous programming models is for mapping onto diverse target architectures. Ideally, the cheapest implementation should use the least strict semantics that meets the requirements of the specific application. Since this knowledge is highly dependent on the application, the choice should be up to the designer.

Our framework for control composition and analysis is based on the *modal process* model. It addresses the problem of control composition by separating the synchronization semantics from the composition, and by supporting automatic synthesis of control communication on distributed architectures. By parameterizing the synchrony of composition, modal processes allow the designer to choose the least expensive synchronization semantics, based on the analysis of our tool.

In this paper, we describe such a tool that aids the designer in analyzing potential problems in the composed system, including deadlock and livelock situations. Analyses for deadlock and livelock conditions are made possible by the explicit constraints that must be stated when composing the states of the processes. In the next section, we review the modal process model, discuss a taxonomy of various semantics, and review the synthesis of control communication. Section 3 describes our analysis algorithm based on the space-time diagram representation.

2 Previous work

2.1 Modal processes

Our model for embedded systems is called modal processes [3]. A process consists of event handlers, data variables, communication ports, and accesses to devices. For the purpose of this discussion, we will focus on the control aspect of the processes. A modal process is a process with modes. A mode is a collection of event handlers. A modal process can have several modes, or several ways of handling events. At any moment, one or more modes may be *active*, though it is also possible that none of the modes is active. After handling an

*This work was supported by DARPA DAAH04-94-G-0272

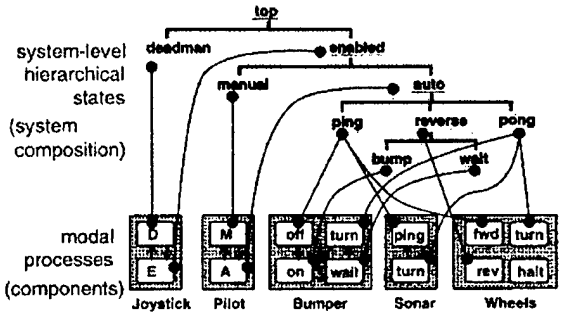


Figure 1: Robot as modal processes with state constraints

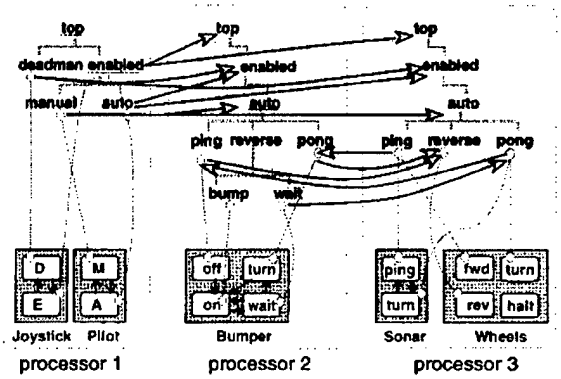


Figure 2: Partitioning the robot onto three processors

event, a handler may also return a value requesting the activation of a new mode.

System composition can be viewed as adding application-specific constraints to the processes. Modes of different processes can be *unified*, which means when one mode is activated in one process, all unified modes in other processes must be activated, even if they are on different processors. These unified modes are said to be bound to a common state. For brevity, we say "activating a state s" to mean "activate the modes unified by state s," as there is no ambiguity. Different states can be related hierarchically, where activating a child state requires activation of its parent. Each state can also constrain its children to be mutually exclusive, or it can allow them to be concurrent – in addition to the process's own constraints on its modes. State de-activation is not expressed explicitly, but is inferred from the constraints. Entering a state requires the deactivation of those states that are mutually exclusive to it. Similarly, exiting a parent state requires the exit of all of its children states.

Fig. 1 shows a robot example described as a composition of modal processes. The robot can be operated with a joystick or can run autonomously with a bumper and a sonar. Their modes are constrained by the state tree above. When the designer partitions the processes onto a distributed architecture, our tool automatically partitions the hierarchical states and synthesizes the control communication accordingly. Fig. 2 shows the robot partitioned onto three processors.

2.2 Taxonomy of control communication

Different semantics can be obtained by changing the synchronization behavior. We have identified the following classes:

Transition synchronous semantics means that every process takes (at most) one-step, or a transition in the Mealy machine sense, in every time step. Examples of this include RTL (register-transfer level) semantics and Esterel.

Event synchronous, a.k.a. *discrete event*, means that all processes see the same set of events simultaneously. Event handling is synchronous to the events, which are totally ordered. It is possible to emit events that are globally visible instantaneously and trigger a potentially infinite chain of transitions within a time step. The time step ends when no more transitions can be triggered. This is the semantics assumed by the original StateCharts as well as the program statements in hardware description languages such as Verilog [9] and VHDL. It is also possible to combine event synchrony with transition synchrony, as in Esterel and the later revised StateCharts [4].

Mode synchronous semantics requires processes to synchronize on a mode change if their modes are affected. Localized transitions that do not affect other processes need no synchronization. Otherwise, the system can have any other combinations of synchrony. This is the semantics assumed by modal processes [3] by default.

In **data synchronous** models, the propagation of control is tied to data communication. Even though a set of processes should *logically* operate in the same modes (as a result of binding), it is unnecessary to synchronize all of them at once on a mode change. Instead, the mode is *correlated* with data and is allowed to be pipelined over data communication channels in the same way data flows through a dataflow network or a pineline. In other words, control can flow synchronously as data, or piggybacked with data. See [8] for a comprehensive survey on dataflow models.

In **asynchronous** models, each process can make arbitrary amounts of progress asynchronously to other processes. Most "process" models fall into this category, most notably CSP [6] and its derivatives. Even synchronous models have come to rely on asynchronous compositions at the system level. For example, CRP [2] is essentially a set of locally-synchronous Esterel components that are composed asynchronously as CSP processes at the system level. StateMate [5] offers similar composition: locally-synchronous StateCharts components are connected together asynchronously in ModuleCharts. Both are motivated by the fact that transition-synchrony and event-synchrony are impractical for distributed systems.

Unfortunately, these models force designers to commit to a specific synchronization semantics at the highest specification level to reflect their architectural mapping concerns. In

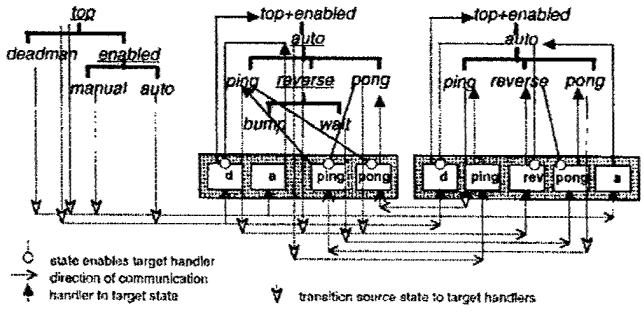
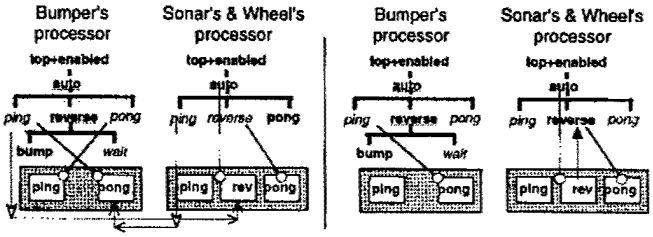


Figure 3: Synthesized control communication and handlers



(a) both Bumper and Sonar communicate mode change to each other.

(b) convergence: Sonar goes to reverse, Bumper denies transition to pong.

Figure 4: Transition scenario

contrast, our modal process model allows the designer to parameterize and synthesize the synchrony that is appropriate for the specific architectural mapping. The advantages are that the modules can be made more retargetable without over-specifying their behavior, and that they enable better design optimization to the target architecture.

2.3 Control communication

To satisfy the mode unification constraints in a distributed architecture, mode changes must be communicated across processors. The run-time system on each processor does not need a full replication of the state space, but only a projection needed for maintaining the local modes and their interactions with the other processes. When a handler initiates a mode change, our run-time system transmits the target-state reference to other processors whose modes are affected, as determined by static analysis. On the receiver's side, we synthesize a handler for each state that can be a remote target.

Fig. 3 shows the synthesized transition handlers in the nested boxes. These target handlers can be qualified by the modes where outgoing transitions to the handlers' targets are actually allowed, though they default to always active in the figure. For example, on the second processor, the target handler for ping is enabled only when in pong state. The ability to deny transition requests selectively is a useful feature for convergence in asynchronous compositions. Fig. 4 shows a scenario of the robot example where two transitions simultaneously requested by both the sonar and the bumper processes result in convergence without any synchronization. The sonar accepts the bumper's request to go to reverse, while the sonar's request to go to pong is dropped while in reverse.

Control communication may need synchronization depending on the composition semantics. Transition-synchronous semantics requires synchronization on every Mealy-machine transition, or every quantum of computation, while event-synchronous semantics requires synchronization on every event, and both are impractical for distributed systems. By default, we assume mode-synchronous semantics, which requires the run-time system to implement the abstraction of always presenting a consistent view of the system state. The run-time system can hide the transient conditions from the components by using a three-phase synchronization protocol on a mode change: the sender sends the request, waits for all receivers to reply, and tells everyone to proceed.

It is possible to further relax the synchronization requirement by presenting individually consistent view on different processors, even though the entire system may not be simultaneously consistent. For example, data-synchronous semantics further eliminates global synchronization by exploiting the regularity in the underlying dataflow model. As long as each process is fired with consistent context, global synchronization is not a requirement. However, this approach requires knowledge about the way control flows through the system. We are currently investigating its application to dataflow models, which have highly regular structures. In this paper, we are mainly concerned with asynchronous composition. It achieves the least expensive implementation for a given target architecture by exposing transient conditions to the user processes. It should be considered if the application can tolerate transient states, especially in areas that are not safety-critical.

3 Analysis

In this section, we describe a method for analyzing the property of a given asynchronous composition. We assume that each mode starts in a consistent *steady state*. A system state is *consistent* if the active modes satisfy the unification, mutual exclusion, and hierarchy constraints. The system is in a (temporary) steady state when all internal (mode-change) events have been generated and handled in response to a given set of input events. Since a given event can occur multiple times, a system may never reach a steady state because the input set is potentially infinite. Here we will use the event inter-arrival constraints to allow us to bound the number of events to consider in a scenario between steady states. Anomalous conditions in asynchronous systems include oscillation, livelock, divergence, and deadlock.

Oscillation means that the system fails to reach a steady state directly, but may be temporarily caught in a cycle of internal events due to race conditions. That is, processes on different processors bounce between different modes trying to unify with each other's modes at the same time telling each other to do otherwise. An oscillation can continue indefinitely and can pose a problem in the ability to satisfy response time

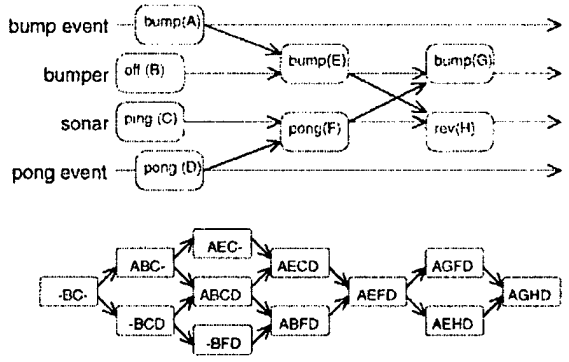


Figure 5: Space-time diagram and lattice of consistent cuts

constraints or even lead to incorrect behavior, although a later event may be able to break the cycle. A stronger case of an oscillation is a livelock, which cannot be broken by any event or race condition.

Note that it is difficult to get into an oscillation and even harder for a livelock because our synthesized run-time system and handlers do not create such a cycle, but anomalous user handlers can. On the other hand, it is easy to reach a steady state, but it is not necessarily correct. If the steady state is inconsistent, we call this *divergence*. A special case of divergence is a deadlock, which is an inconsistent steady state that responds to no events and therefore the system has no way of exiting. It is possible for a consistent steady state to respond to no events, but it would not be considered a deadlock here.

3.1 Space-time diagram

The representation for our analysis is based on the space-time diagram as used by Lamport [7] for describing events in distributed systems. The diagram contains vertices that are divided into parallel tracks, each of which models a physically separate module with its own views of time and event arrivals. Event occurrences are represented by vertices, and they are connected by directed edges that represent "happens-before" relations. Note that the term "event" in a space-time diagram is more abstract and general than those in a behavioral description because it marks not only I/O and internal communication but also mode changes. These events are totally ordered on a given track but partially ordered between different tracks. The happens-before edges capture causality due to sequencing and communication.

The instantaneous state of a system as modeled by a space-time diagram can be characterized by dividing the events into two sets: a past set and a future set. A cut that has no edges going from the future to the past in a space-time diagram is called a *consistent cut*. There can be many consistent cuts in a given space-time diagram representing different points in a trace. Different consistent cuts of a given diagram can be partially related to form a graph (specifically, a lattice). It will be used as our primary representation for analysis. Fig. 5 shows a space-time diagram and the corresponding lattice for the transition scenario in Fig. 4. A lattice can be constructed by traversing the space-time diagram. The label for the new vertex in the lattice is the concatenation the labels of all events on the current cut. Initially, the past set is the precondition and the future set contains the entire vertex set. We create a new vertex by moving an event from the future set to the past set if all of its predecessors are in the past set.

3.2 Formulation and algorithm

Our approach to the deadlock/livelock analysis is to analyze the causality graph, which is based on the graph of consistent-cuts. An oscillation/livelock analysis involves cycle detection in the causality graph. If it is acyclic, then we check for divergence by inspecting the states encoded by the sink vertex.

To construct the causality graph, we need to encode causality and the event's class using the event labels. The *causal set* of a given event instance is the union of all of those events on its incoming edges. To encode causality in an event label, we concatenate the class label of the event itself with the class labels of those events in its causal set. We modify the routine for allocating a new vertex for a consistent cut. The new allocation routine must return the same vertex every time if it is called with the same causality. A cycle is detected when the routine returns the same vertex twice.

The analysis algorithm constructs causality graphs for all possible event orders and for all reachable steady states. For every consistent-cut graph, the algorithm continues growing the graph until a cycle is detected or when no more vertex can be added. A cycle indicates a potential livelock. If the graph has no cycles, we check the consistency of the sink vertex by substituting the active modes into the constraint equations, which are captured by the mode bindings and the state hierarchy.

3.3 Examples

The example shown in Fig. 5 is an analysis of a potential race condition in Fig. 4. The bumper module is requesting a mode change to reverse while the sonar detects an obstacle and requests a transition to turn mode. We obtain a finite space-time diagram and an acyclic causality graph. Therefore, we can verify the sink vertex for state consistency. The bumper's processor is in bump state while the sonar's processor is in reverse. Because bump is a child state of reverse, the system converges to a consistent state for this scenario. If we continue enumerating all possible orders of event observation and conditional branches, we will find that they converge to the same state. Therefore, asynchronous composition can be considered for implementation.

A subtle modification to the bumper process would introduce divergence. For example, suppose the bumper process contains a transition edge from sf bump to pong. The node labeled G in Fig. 5 would represent a transition from to pong state, instead of remaining in bump. Steady state analysis shows that pong and bump are simultaneously active on different processors even though they should be mutually exclusive. Therefore, an asynchronous implementation can lead

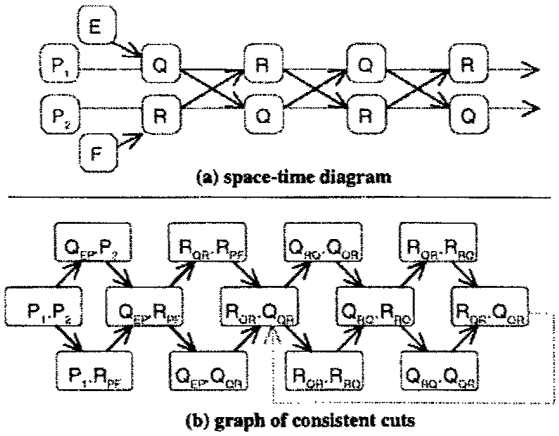


Figure 6: Livelock example

to inconsistent behavior. The problem can be fixed by mode-synchronous composition, where the run-time system resolves the conflicting transition requests before returning control to user code.

Real livelocks involve very intricate sequences of events that are too complex to explain for this paper. Instead, we use the simplest possible, though rather hypothetical, scenario as shown in Fig. 6 to illustrate the concepts. Even this seemly trivial example involves over a dozen vertices for consistent cuts. The reason for the complexity is that even though the same modes are visited repeatedly, the causality does not repeat until much later. The earlier vertices are caused by external events, while the repeated vertex is in a causality cycle without external events.

4 Conclusions

Many control-dominated specification models have relied on strict synchrony to give them well-defined composition semantics and determinism. Unfortunately, strict synchrony makes them impractical for targeting heterogeneous, distributed architectures. In this paper, we presented an evaluation method that helps designers determine if alternative composition synchrony yields the same behavior when transient conditions can be tolerated. Such consistency tests and implementation freedom are made possible by the specification model, namely modal processes, where the control states of the processes are explicitly bound and constrained independently of their synchrony. It allows the designer to parameterize, rather than dictates, the synchrony needed for their specific application. The goal of our tool is to help the designer achieve the least expensive implementation by synthesizing the corresponding run-time support and perform analysis for their evaluation. This paper addresses the asynchronous option; we are currently investigating data-synchronous composition and other alternatives.

Currently, manual designs involve mostly asynchronous composition for practical reasons, but designers are burdened with the task of analyzing race conditions in terms of low level primitives like semaphores. It is extremely error-prone and difficult to cover all possible intricate cases, and we have presented a systematic approach to automating this task. Although it can be expensive for computers to exhaustively enumerate all possible states, many cases can be pruned in practice. Modal processes constrain the state space by unifying states across different modules, so that many combinations can be ruled out. Also, race conditions are possible only when multiple processes request conflicting mode changes simultaneously, and these cases are relatively straightforward to identify. We believe that the analysis tool will help designers identify more system-level optimization opportunities without resorting to ad-hoc techniques that hinder design maintainability and retargetability.

References

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, January 1993.

[3] P. Chou and G. Borriello. Modal processes: Towards enhanced retargetability through control composition of distributed embedded systems. In *Working paper, http://www.cs.washington.edu/research/chinook/publications.html*, October 1997.

[4] D. Harel and E. Gery. Executable object modeling with StateCharts. *ICSE-18*, pages 246–257, 1996.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[8] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83:773–801, May 1995.

[9] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, 1991.